Нейронные сети и обучение с подкреплением в играх с полной информацией
Neural networks and reinforcement learning in games with complete information

Москва, 2019 г.

Содержание

1 Введение

Создание компьютерных программ для решения игр с полной информацией получило свое начало с работы Алана Тьюринга в 1948 году.

Шашки, Шахматы, Реверси, Го, Крестики-нолики, Кубик и многие другие относятся к играм с полной информацией. Т.е. оба игрока знают все о своей позиции и вариантах доступных им ходов. В таких играх можно определить своего рода функцию, которая для любой выбранной позиции на игровой доске возвращает оптимальный ход для игрока при условии оптимальной игры всех сторон. Иметь такую функцию $\hat{X}$ будет означать математическое решение игры (поиск глобального оптимума).

Изучение игр с полной информацией идет уже давно. Тьюринг, Шеннон, Нейман разработали аппаратное обеспечение, алгоритмы и теорию для анализа игр. Впоследствии игры стали главной задачей для поколения исследователей искусственного интеллекта, кульминацией которых стали высокопроизводительные компьютерные программы, превосходящие сверхчеловеческий уровень. Однако, эти системы сильно настроены на свою область и не могут быть обобщены на какую-либо другую игру без значительных вмешательств человека.

Игра с полной информацией являются наиболее широко изученной областью в истории искусственного интеллекта. Самые сильные программы основаны на сочетании сложны методов поиска, игровых адаптаций, функций оценки (разработанных вручную экспертами в течении нескольких десятилетий). В отличие от программы *AlphaGo Zero*, недавно добившейся сверхчеловеческих результатов в игре **GO**, благодаря обучению с подкреплением из самостоятельных игр. В этой работе обобщается этот подход в единый алгоритм *Alpha Zero*, который может достичь высокой производительности в нескольких играх с полной информацией, без каких либо дополнительных знаний, кроме правил игры.

В данной работе предлагается рассмотреть версию искусственного интеллекта **Alpha Zero**, основанный на сверточных нейронных сетях с подкреплением в играх с полной информацией. И применение ИИ на основе игрового фреймворка [4] в игре - *3D-крестики-нолики*.

2 Постановка задачи

Практически все игры можно представить в виде тернарных деревьев решений, где каждый узел будет представлять собой один шаг решения задачи (ход в игре). Ветвь в дереве соответствует решению, которое ведёт к более полному выводу. Листы представляют собой окончательное решение (итоговые позиции). Наша цель – найти в дереве лучший путь от корня до листа.

Деревья решений обычно невероятно велики. Например, в игре крестики-нолики дерево содержит более полумиллиона узлов. Понятно, что в точках, шашках, шахматах, Go и других играх деревья на порядок больше.

Шашки, Шахматы, Реверси, Го, Крестики-нолики, Точки и многие другие относятся к играм с полной информацией. Т.е. оба игрока знают все о своей позиции и вариантах доступных им ходов. В таких играх можно определить своего рода функцию, которая для любой выбранной позиции на игровой доске возвращает оптимальный

ход для игрока при условии оптимальной игры всех сторон. Иметь такую функцию $\hat{X}$ будет означать математическое решение игры (поиск глобального оптимума).

Может показаться, что создание такой функции достаточно просто: перебрать все дерево вариантов, которые доступны игрокам. Но тут и появляется первая проблема - в приличных играх вариантов развития событий слишком много для простого перебора.

Однако были придуманы несколько методов замены функции $\hat{X}$ на ее приблизительный аналог $X \approx \hat{X}$, который можно вычислить уже за какое-то разумное время. Один из самых очевидных и простых способов решения этой задачи - подрезка "плохих"ветвей дерева перебора. Идея здесь в том, что в игре обычно существуют позиции, которые либо "очевидно плохие либо "очевидно хорошие". В тех же крестиках-ноликах, позиция на недоигранной доске может быть "очевидно плоха"для одной из сторон. Достижение такой позиции уже можно считать проигрышем - не вдаваясь в подробности, как это произойдет, если доводить игру до конца. Такой подход позволяет сократить глубину перебора дерева. Так же существует вариант, основанный на сокращении ширины перебора, т.е. можно перебирать не все возможные варианты ходов, а только из набора наиболее популярных - на основе записей известных партий. Такой подход позволяет сократить число вариантов развития событий и время вычисления. Но игра такой программы может стать предсказуемой, а для больших игр может найтись уникальная расстановка для которой не будет найдено решения в базе.

На данный момент одним из главных подходов при создании алгоритмов в играх, для которых существует полная информация, но не возможен исчерпывающий перебор, является метод иерархического поиска Монте-Карло.

Вместо альфа-бета-поиска с улучшениями, специфичными для предметной области, AlphaZero использует универсальный алгоритм поиска по дереву Монте-Карло (MCTS).

Алгоритм Alpha Zero, описанный в этой работе, отличается от оригинального AlphaGo алгоритма в нескольких отношениях. AlphaGo оценивает и оптимизирует вероятность выигрыша, принимая бинарные исходы - выигрыш / проигрыш. Вместо этого Alpha Zero оценивает и оптимизирует ожидаемый результат, принимая во внимание ничьи или потенциально другие результаты.

Рассмотрим:

- Версию алгоритма Монте-Карло(MCTS) под названием Upper Confidence bound applied to Trees (UCT).
- Применение алгоритма MCTS в сверточных нейронных сетях.
- Применение на антогонистической игре с полной информацией (3D-Крестики-нолики).

Так же стоит задача в создании антогонистической игры на основе игрового *фреймворка* [4] с возможностью играть против человека или компьютера (алгоритма поиска нуилучшего решения).

3 Алгоритм

3.1 Поиск по дереву методом Монте-Карло

Поиск по дереву методом Монте-Карло относится к семейству древовидных алгоритмов поиска. Появился в 2006 году. Новшеством поиска по дереву методом Монте-Карло является использование случайных симуляций, как оценки для значений хода в совокупности с деревом поиска, чтобы сбалансировать использование многообещающих ходов с исследованием непроверенных ходов. Практически сразу MCTS привел к большим достижениям в области компьютерных Go, шахмат и др. MCTS в основном применяется в играх с полной информацией. Это неудивительно, учитывая, что алгоритм UCT(Upper Confidence bound applied to Trees) сходится к минимаксному решению в этих играх.

3.2 Описание алгоритма MCTS

До сих пор главным подходом к созданию игровых алгоритмов в играх, где открыта информация, но невозможен исчерпывающий перебор, были так называемые методы иерархического поиска Монте-Карло (MCTS). Работают они следующим образом. Представьте, что Вы переехали в новый город и хотите выбрать для себя хороший книжный магазин. Вы заходите в один случайно выбранный магазин, подходите к случайной полке, берете случайную книгу, открываете на случайном месте и читаете, что вам попалось на глаза. Если это хороший текст, магазин в ваших глазах получает плюс к карме, если нет — минус. Проведя несколько таких забегов можно обнаружить, что в одном из магазинов вы часто наталкиваетесь на слова вроде «дети» или «сладости», а в другом более популярны «нелокальность» или «алгоритм». Постепенно становится понятно, какой из книжных вам больше подходит.

Методы поиска Монте-Карло работают по такому же принципу: из данной позиции s проводится симуляция игры до самого победного (или проигрышного) конца, причем каждый ход на каждом шаге игры выбирается случайным образом. Таким образом, проведя множество случайных симуляций можно грубо оценить выгодность позиции без исчерпывающего перебора. При этом, конечно, есть вероятность пропустить среди множества проигрышных и победные варианты, но эта вероятность уменьшается с увеличением числа симуляций. Оценочная функция $V(s)$, полученная таким перебором, асимптотически приближается к истинной $V*(s)$ полного перебора.

3.3 Алгоритм поиска по дереву методом Монте-Карло

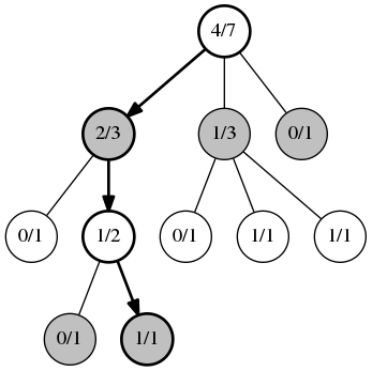
Существует несколько различных вариаций MCTS алгоритмов, однако все они имеют общую структуру. Алгоритм выполняет большое количество симуляций игры и строит частичное дерево, добавляя по одному узлу на каждой итерации. MCTS принимает решение на основе результатов моделирования, которые хранятся в этом дереве. Каждая смоделированная игра называется итерацией алгоритма MCTS и состоит из следующих 4-х шагов:

- Выбор: выбираются ходы, которые уравновешивают использование хороших ходов и исследование непроверенных ходов, пока не будет достигнуто состояние,

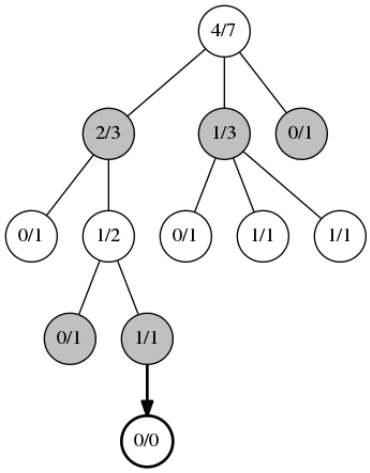
которое не представлено в дереве MCTS.

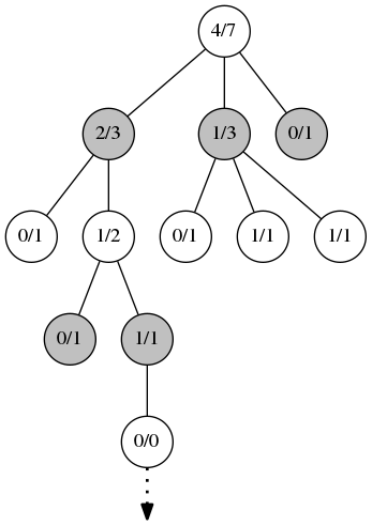
- Расширение: Добавление нового узла в дерево, соответсвующее непосещенному состоянию, обнаруженному во время выбора.
- Симуляция: разыгрывать остальную часть игры, используя случайную политику.
- Обратное распространение: обновление узлов в дереве с результатом игры, полученном на этапе симуляции.

Основная сложность в выборе дочерних узлов заключается в поддержании некоторого баланса между использованием вариантов ходов с высокой средней вероятностью выигрыша и исследованием ходов с небольшим количеством симуляций. Рассмотрим версию алгоритма Монте-Карло под названием Upper Confidence bound applied to Trees (UCT)(3.7).Основой алгоритма UCT является решение задачи многоруких бандитов. В частности используется алгоритм Upper-Confidence-Bound (UCB). Рассмотрим пошагово каждую фазу алгоритма:



- Первая фаза, **выбор**. Каждую позицию мы рассматриваем как задачу многорукого бандита. Узлы на каждом этапе выбираются согласно алгоритму UCB. Эта фаза действует до тех пор, пока не будет найден узел в котором еще не все дочерние узлы имеют статистику побед. На рисунке первое значение в узле это количество побед, второе общее количество игр в этом узле.
- Вторая фаза,**расширение**. Когда алгоритм UCB больше не может быть применим, добавляется новый дочерний узел.

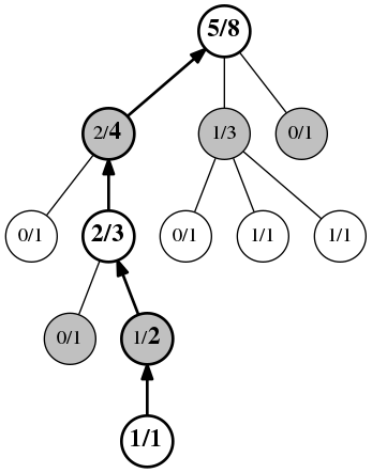




- Третья фаза, **симуляция**. Из созданного на предыдущем этапе узла запускается игра со случайными или, в случае использования эвристик, не совсем случайными ходами. Игра идет до конца партии. Здесь важна только информация о победителе, оценка позиции не имеет значения.

- Четвертая фаза, **обратное распространение**. На этом этапе информация о сыгранной партии распространяется вверх по дереву, обновляя информацию в каждом из ранее пройденных узлов. Каждый из этих узлов увеличивает показатель количества игр, а узлы, совпадающие с победителем, увеличивают также и количество побед. В конце алгоритма выбирается узел, посещенный наибольшее количество раз.

UCT не всегда выбирает только самый лучший ход, но так же периодически исследует и менее успешные узлы. Второй параметр формулы медленно растет для тех узлов, которые посещаются не так часто. И в итоге на каком-то этапе алгоритм выберет именно такой узел в качестве предпочтительного. Если узел оправдал ожидания, в следующий раз он будет выбран с большей вероятностью.



3.4 Свойства

MCTS имеет несколько хороших свойств по сравнению с другими алгоритмами поиска. Во-первых, MCTS может быть применен к любой задаче, которая моделируется по решению в виде дерева и может применяться без каких либо областных знаний (хотя дополнительные знания в области дают преимущества). Поэтому MCTS можно охарактеризовать как эвристический метод. В отличии от популярного минимаксного поиска, где качество игры существенно зависит от эвристики, используемой для оценки возможных исходов не листовых узлах. Однако, в случаях, когда факторы ветвления представляют собой величины больших порядков и трудно найти хорошие эвристики, производительность минимакса значительно ухудшается. MCTS также является *anytime* алгоритмом, так как его можно остановить в любомй момент и принять решение, даже если алгоритм прерван до его завершения. Это отличие от алгоритмов, таких как *минимаксный поиск*, который должен завершить вычисления, прежде чем решение может быть принято. Наконец, MCTS выполняет ассиметричный поиск, а не поиск на фиксированную глубину. Это позволяет тратить больше времени на поиск более лучших ветвей игры и меньше времени на поиск областей дерева с неоптимальным.

3.5 Задача о многоруком бандите

В задаче о многоруком бандите у игрока есть выбор из нескольких игровых автоматов, каждый из которых имеет различную вероятность выплаты. Цель задачи состоит в том, чтобы сбалансировать трату времени на оценку того, какая машина имеет большую вероятность выигрыша и тем самым тратить время преимущественно на лучшие игровые автоматы. Многорукий бандит имеет конечное число рук, каждая с неизвестной наградой распределения. Задача в том, чтобы определить политику, которая будет принимать в качестве входных данных все награды, полученные на предыдущих испытания и определять, какая рука должна быть выбрана следующей. *Regret* - термин, используемый для описания потерь, понесенных при игре неоптимальной рукой, котороый представляет собой разницу между полученной наградой и ожидаемого вознаграждения от оптимальной руки. Совокупность *regret* является суммой *regret*, понесенных во всех предыдущих испытаниях. Стандартный способ оценки выбора политики в задачи многорукого бандита заключается в определении ожидаемого значения суммарного *regret* после определенного количества испытаний. В случае политики UCB, ожидаемое суммарное *regret* растет как логарифм от числа последовательных испытаний. Задача выбора применяется в MCTS. Задачей выбора действия в игровом дереве является своего рода задаче многорукого бандита. В этом случае руки соответствуют различным действиям, а награды - результаты симуляций игры. Алгоритм UCT(Upper Confidence bound applied to Trees) строит частичное игровое дерерво итеративно и алгоритм UCB используется для выбора действий внутри дерева(а также симуляции Монте-Карло используются для выбора действий из состояний вне дерева). Для игры с полной информацией, оптимальным действием будет действие с наибольшим минимаксным значением и алгоритм UCT сходится к этому значению по мере увеличения числа итераций.

3.6 Задача о K-руком бандите

Задача о бандите, описанная ранее, также известна как задача о K-руком бандите. Задача определяется следующим образом:

Определение 3.1. *Задача K-рукого бандита состоит из случайных переменных* $X_{i,n}$ *для* $1 \leq i \leq k$ *и* $n \geq 1$. *Это соответствует рукам на* K *игровых автоматах, где* I *обозначает индекс каждой руки и* N *обозначает награды, полученные на* N*-ой игре соотвествующей руки. Кроме того, случайные величины* $X_{i,n}$ *удовлетворяют следующим свойствам:*

- *Случайные величины* $X_{i,n}$ *для* $n \geq 0$ *являются независимыми и одинаково распределены, согласно некоторому распределению с математическим ожиданием* $E[X_{i,n}] = \mu_i$.
- *Случайные величины* $X_{i,s}$ *и* $X_{j,t}$ *независимы, но не обязательно одинаково распределены для* $1 \leq i < j \leq k$ *и* $s, t \geq 1$

т.е. руки могут иметь различные распределения вознаграждений, но распределение для данной руки не меняется со временем. Термин испытание будем использовать для описания процесса выбора руки для игры и затем получать награду. Сколько раз рука была сыграна во время первых n испытаний, обозначим через $T_i(n)$.

3.7 Алгоритм задачи о многоруком бандите

Существует много различных способов для решения задачи многорукого бандита, также известного как бандитский алгоритм. Алгоритм решает, на какую руку брать заданную итерацию, учитывая историю предыдущих розыгрышей и наград. В нашем случае UCB - алгоритм, используемый для выбора действий в MCTS, который выбирает руку с максимальным

$$\overline{X_{i,n}} + C\sqrt{\frac{\log n}{T_i(n)}}$$

где $\overline{X_{i,n}}$ - средняя награда, полученная после $T_i(n)$ испытаний i-ой руки(из n испытаний в сумме) и C - числовая константа. UCB - направленный алгоритм, используется в некоторых приложениях MCTS, устроняет необходимость настраивать константу C и часто приводит к лучшей производительности.

На практике для поиска в дереве ходов настольных игр часто используется модификация формулы UCB. Например, такая:

$$\frac{w_i}{T_i(n)} + c\sqrt{\frac{\log n}{T_i(n)}}$$

здесь w_i это количество побед i-го узла. $T_i(n)$ — количество посещений i-го узла, а n - количество посещений всех соседних узлов. c - константа, используемая для установки нужного баланса между шириной и глубиной поиска. Чем она больше, тем более глубокий будет поиск. Первый компонент формулы соответствует эксплуатации; это повышает значение для ходов с высоким средним коэффициентом выигрыша. Второй компонент соответствует разведке; это повышает значение для ходов с небольшим количеством симуляций.

3.8 Применение в нейронных сетях

Представим общий обзор используемого алгоритма, который основывается на AlphaGo Zero [9]. Алгоритм основан на самостоятельной игре и не использует никаких человеческих знаний, кроме правил игры. В основе используется нейронная сеть, которая оценивает значения данного состояния доски и оптимальную политику. Самостоятельная игра руководствуется методом поиска по дереву Монте-Карло(MCTS), который действует как оператор по улучшению политики. Результаты каждой игры затем используются в качестве наград, которые используются для обучения нейронной сети с улучшенной политикой. Обучение выполняется в итеративной форме - текущая сеть используется для выполнения самостоятельной игры, результаты которой будут затем использоваться для перетренировки сети.

3.8.1 Нейронная политика и сети

Используем нейронную сеть f_θ с параметром θ, которая принимает в качестве ввода состояния доски s и выдает следующее значение состояния доски $v_\theta \in [-1; 1]$ для текущего игрока и вектор вероятностей $\vec{p}$ по всем возможным действиям. $\vec{p_\theta}$

представляет собой стохастическую политику, которая используется для самостоятельной игры. Нейронная сеть воспроизводится случайным образом. В конце каждой итерации самостоятельной игры, нейронная сеть выдает примеры обучения в форме $(s_t, \vec{\pi_t}, z_t)$. Где $\vec{\pi_t}$ дает улучшенную оценку политики после выполнения MCTS, начиная с позиции s_t, и $z_t \in \{-1; 1\}$ - окончательный результат игры для текущего игрока. Нейронная сеть затем обучается минимизировать следующую функцию потерь:

$$l = \sum_t (v_\theta(s_t) - z_t)^2 + \vec{\pi_t} \log(\vec{p_\theta}(s_t))$$

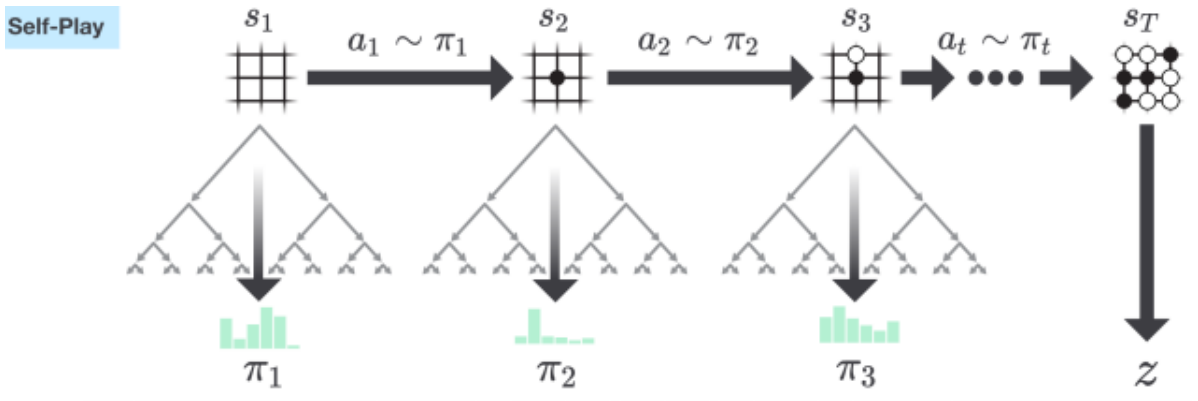


Рис. 1: Самостоятельная игра

Используем нейронную сеть, которая берет необработанные состояния доски в качестве входных данных. Затем следует 4 сверточные сети с двумя полносвязными сетями прямого распространения. Далее следует два соединительных слоя - один выводит v_θ, второй выводит вектор $\vec{p_\theta}$. Обучение выполняется с использованием *AdamOptimizer* [10] и *BatchNormalisation* [11].

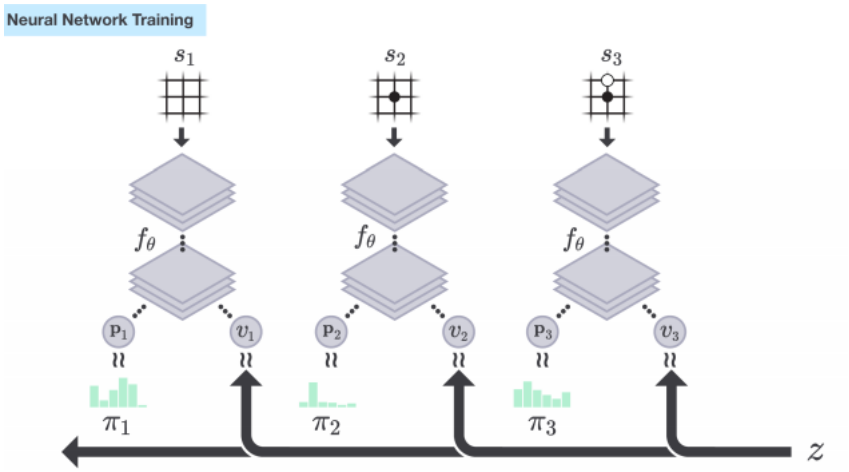


Рис. 2: Тренировка нейронной сети

3.8.2 MCTS для улучшения политик

Используем MCTS для улучшния политик, изученных нейронной сетью. MCTS исследует дерево, в котором узлы представляют собой различные состояния доски, а направленные ребра - переходы между двумя узлами $(i \to j)$, если действие допустимо. Начиная с пустого дерева поиска, мы расширяем его по одному узлу(состоянию) за раз. При обнаружении нового узла, значение получается из самой нейронной сети.

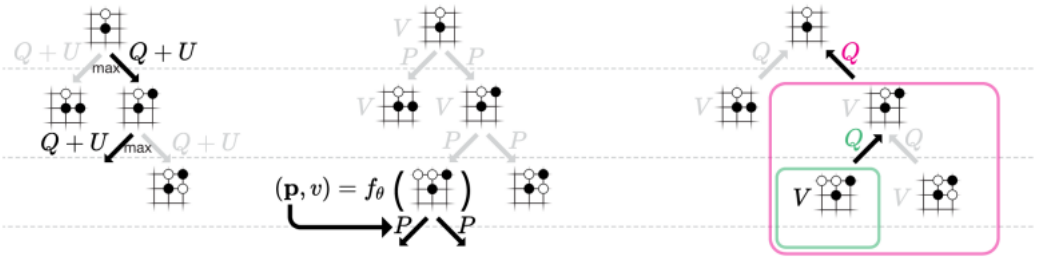
Для поиска по дереву мы используем следующие значения:

- $Q(s,a)$ - ожидаемое вознаграждение после принятий действий a из состояния доски s.
- $N(s,a)$ - Количество принятий действий a из состояния доски s. (Т.е. количество симуляций из узла s).
- $P(s,\cdot) = \vec{p_\theta}$ - начальная оценка принятия действий из состояния s в соотвествии с политикой, возвращаемой текущей нейронной сетью.

Используя это, мы можем вычислить $U(s,a)$ (UCB) для значения Q следующим образом:

$$U(s,a) = Q(s,a) + c \cdot P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)}$$

Чтобы использовать MCTS для улучшения начальной политики, возвращаемой текущей нейронной сетью, инициализируем пустое дерево, выбрав s как корень. Однократная симуляция выполняется следующим образом. Выполним действие a, чтобы максимизировать UCB $U(s,a)$. Если следующее состояние s'(полученное после действия a из состояния s) существует, мы рекурсивно вызываем поиск из s'. Если состояние не существует, добавляем новое состояние в наше дерево, инициализируем $P(s',\cdot) = \vec{p_\theta}(s')$ и значение $v(s') = v_\theta(s')$ для нейронной сети, а так же $Q(s',a)$ и $N(s',a)$. Затем мы распространяем $v(s')$ вверх по пути текущей симуляции и обновляем все $Q(s,a)$. Однако, если мы достигли терминального состояния игры, мы распространяем фактическое вознаграждение (+1 , если игрок выиграл, иначе -1).



После нескольких симуляций, значение $N(s,a)$ в корне обеспечивает лучшее приближение для политики. Улучшенная стохастическая политика $\vec{\pi}(s)$ - это просто нормализованное значение $\frac{N(s,\cdot)}{\sum_b (N(s,b))}$. Во время самостоятельной игры выполняем MCTS и выбираем ход из улучшенной политики $\vec{\pi}(s)$. Ниже приведем реализацию одной симуляции алгоритма поиска.4.1

3.8.3 Политика итераций во время самостоятельной игры

Обучение через самостоятельную игру является алгоритмом итерации политики - мы играем в игры и вычисляем значения Q, используя нашу текущую политику симуляций(в данном случае нейронную сеть), а затем обновляем все значения, используя вычисленную статистику.

Полный алгоритм обучения.4.2 Мы инициализируем нашу нейронную сеть со случайными весами. На каждой итерации алгоритма мы воспроизводим несколько самостоятельных игр. На кажом ходу игры, выполняем фиксированное количество симуляций MCTS из состояния s_t. Выбираем ход из улучшеных политик $\vec{\pi_t}$. Это дает нам обучающий пример (s_t, π_t, z). Награда z заполняется в конце игры: $+1$ - победа, -1 - поражение. См. рис.1

В конце итераций, нейронная сеть обучается на полученных обучающих примерах. Старые и новые сети противостоят друг другу. Если новая сеть выигрывает у старой больше игр, чем установленный порог (55%), то сеть обновляется до новой. В противном же случае, мы проводим еще одну итерацию, чтобы дополнить обучающие примеры. Код для полгоного алгоритма обучения приведен ниже 4.2.

4 Реализация

4.1 Одна симуляция алгоритма поиска

```
def search(s, game, nnet):
    if game.gameEnded(s): return -game.gameReward(s)

    if s not in visited:
        visited.add(s)
        P[s], v = nnet.predict(s)
        return -v

    max_u, best_a = -float("inf"), -1
    for a in range(game.getValidActions(s)):
        u = Q[s][a] + c_puct*P[s][a]*sqrt(sum(N[s]))/(1+N[s][a])
        if u>max_u:
            max_u = u
            best_a = a
    a = best_a

    sp = game.nextState(s, a)
    v = search(sp, game, nnet)

    Q[s][a] = (N[s][a]*Q[s][a] + v)/(N[s][a]+1)
    N[s][a] += 1
    return -v
```

Замечание: Возвращаемые значения являются отрицательными значениями текущего состояния. Это так, потому что величина v принадлежит отрезку $[-1, 1]$, и если v - это величина для состояния текущего игрока, то величиной для состояния другого игрока будет $-v$, поскольку игра антогонистическая.

4.2 Алгоритм обучения

```
def policyIterSP(game):
    nnet = initNNet()         # инициализация нейронной сети
    examples = []
    for i in range(numIters):
        for e in range(numEps):
            examples += executeEpisode(game, nnet)  # собираем примеры из этой игры
        new_nnet = trainNNet(examples)
        frac_win = pit(new_nnet, nnet)    # сравнение новой и прошлой сетей
        if frac_win > threshold:
            nnet = new_nnet               # Заменяем сеть
    return nnet

def executeEpisode(game, nnet):
    examples = []
    s = game.startState()
    mcts = MCTS()                   # инициализируем дерево поиска

    while True:
        for _ in range(numMCTSSims):
            mcts.search(s, game, nnet)
        examples.append([s, mcts.pi(s), None])     # награды пока не могут быть опред
        a = random.choice(len(mcts.pi(s)), p=mcts.pi(s))    # действие из улучшенной
        s = game.nextState(s,a)
        if game.gameEnded(s):
            examples = assignRewards(examples, game.gameReward(s))
            return examples
```

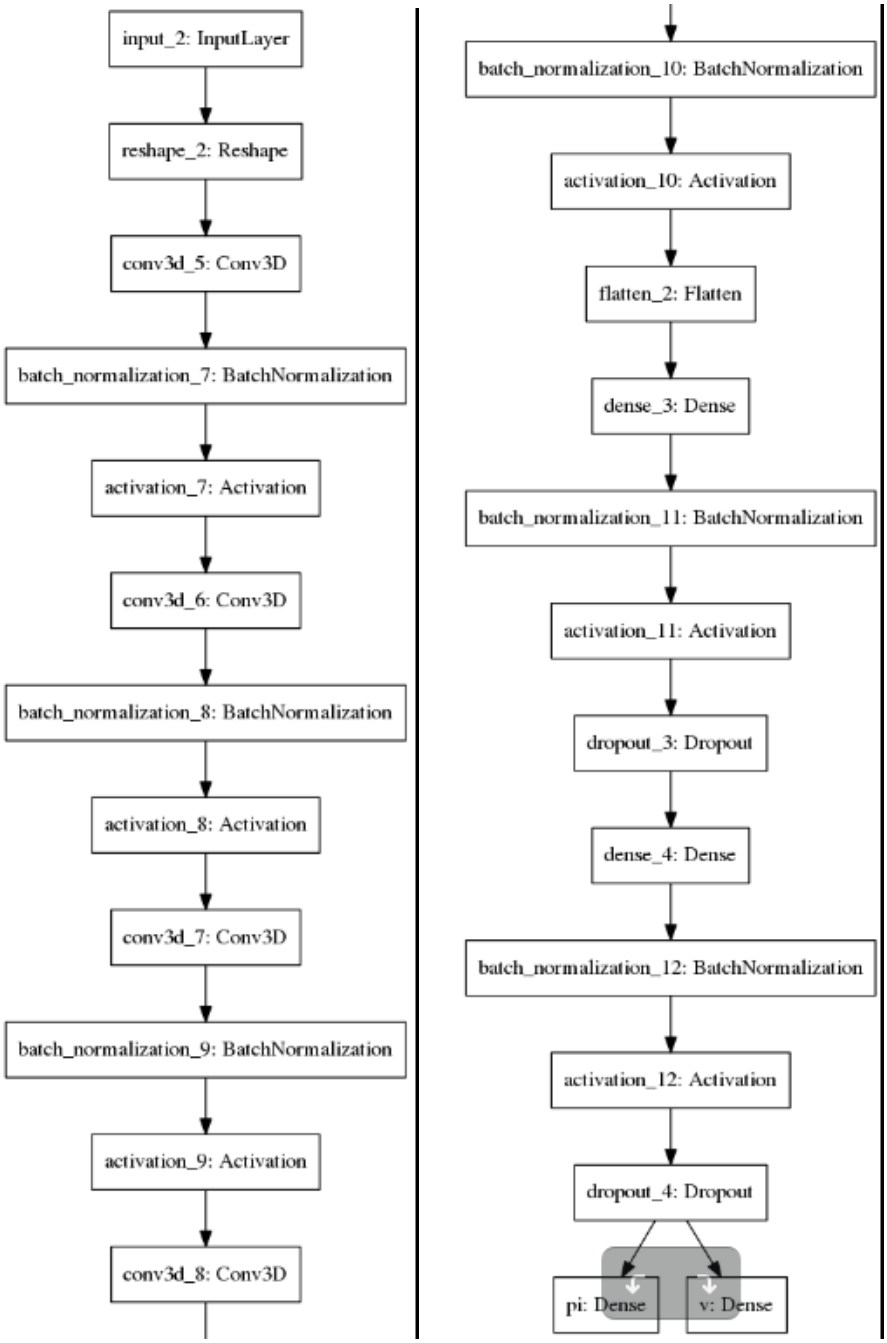
4.3 Архитектура нейронной сети

Версия архитектуры нейронной сети, основанной на фреймворке *keras* (tensorflow).

```
# game params
board_z, board_y, board_x = game.getBoardSize()
action_size = game.getActionSize()
args = args #некоторая настройка сети

# Neural Net
input_boards = Input(shape=(board_z*board_y, board_x))

x_image = Reshape((self.board_z, self.board_y, self.board_x, 1))(self.input_boards)
h_conv1 = Activation('relu')(BatchNormalization(axis=4)
         (Conv3D(args.num_channels, 3, padding='same',use_bias=False)(x_image)))
h_conv2 = Activation('relu')(BatchNormalization(axis=4)
         (Conv3D(args.num_channels, 3, padding='same',use_bias=False)(h_conv1)))
h_conv3 = Activation('relu')(BatchNormalization(axis=4)
         (Conv3D(args.num_channels, 3, padding='same',use_bias=False)(h_conv2)))
h_conv4 = Activation('relu')(BatchNormalization(axis=4)
         (Conv3D(args.num_channels, 3, padding='valid',use_bias=False)(h_conv3)))
h_conv4_flat = Flatten()(h_conv4)
s_fc1 = Dropout(args.dropout)(Activation('relu')
         (BatchNormalization(axis=1)(Dense(1024)(h_conv4_flat))))
s_fc2 = Dropout(args.dropout)(Activation('relu')
         (BatchNormalization(axis=1)(Dense(512)(s_fc1))))
pi = Dense(action_size, activation='softmax', name='pi')(s_fc2)
v = Dense(1, activation='tanh', name='v')(s_fc2)

model = Model(inputs=input_boards, outputs=[pi, v])
model.compile(loss=['categorical_crossentropy',
                        'mean_squared_error'], optimizer=Adam(args.lr))
```

4.4 Визуализация графа нейронной сети

Граф нейронной сети, используемой для обучения антогонистической игры (3d-Крестики-нолики)



Визуализация нейронной сети:

```
Layer (type)                 Output Shape              Param #   Connected to
==================================================================================================
input_2 (InputLayer)         (None, 4, 4, 4)           0
__________________________________________________________________________________________________
reshape_2 (Reshape)          (None, 4, 4, 4, 1)        0         input_2[0][0]
__________________________________________________________________________________________________
conv3d_5 (Conv3D)            (None, 4, 4, 4, 512)      13824     reshape_2[0][0]
__________________________________________________________________________________________________
batch_normalization_7 (BatchNor (None, 4, 4, 4, 512)   2048      conv3d_5[0][0]
__________________________________________________________________________________________________
activation_7 (Activation)    (None, 4, 4, 4, 512)      0         batch_normalization_7[0][0]
__________________________________________________________________________________________________
conv3d_6 (Conv3D)            (None, 4, 4, 4, 512)      7077888   activation_7[0][0]
__________________________________________________________________________________________________
batch_normalization_8 (BatchNor (None, 4, 4, 4, 512)   2048      conv3d_6[0][0]
__________________________________________________________________________________________________
activation_8 (Activation)    (None, 4, 4, 4, 512)      0         batch_normalization_8[0][0]
__________________________________________________________________________________________________
conv3d_7 (Conv3D)            (None, 4, 4, 4, 512)      7077888   activation_8[0][0]
__________________________________________________________________________________________________
batch_normalization_9 (BatchNor (None, 4, 4, 4, 512)   2048      conv3d_7[0][0]
__________________________________________________________________________________________________
activation_9 (Activation)    (None, 4, 4, 4, 512)      0         batch_normalization_9[0][0]
__________________________________________________________________________________________________
conv3d_8 (Conv3D)            (None, 2, 2, 2, 512)      7077888   activation_9[0][0]
__________________________________________________________________________________________________
batch_normalization_10 (BatchNo (None, 2, 2, 2, 512)   2048      conv3d_8[0][0]
__________________________________________________________________________________________________
activation_10 (Activation)   (None, 2, 2, 2, 512)      0         batch_normalization_10[0][0]
__________________________________________________________________________________________________
flatten_2 (Flatten)          (None, 4096)              0         activation_10[0][0]
__________________________________________________________________________________________________
dense_3 (Dense)              (None, 1024)              4194304   flatten_2[0][0]
__________________________________________________________________________________________________
batch_normalization_11 (BatchNo (None, 1024)           4096      dense_3[0][0]
__________________________________________________________________________________________________
activation_11 (Activation)   (None, 1024)              0         batch_normalization_11[0][0]
__________________________________________________________________________________________________
dropout_3 (Dropout)          (None, 1024)              0         activation_11[0][0]
__________________________________________________________________________________________________
dense_4 (Dense)              (None, 512)               524288    dropout_3[0][0]
__________________________________________________________________________________________________
batch_normalization_12 (BatchNo (None, 512)            2048      dense_4[0][0]
__________________________________________________________________________________________________
activation_12 (Activation)   (None, 512)               0         batch_normalization_12[0][0]
__________________________________________________________________________________________________
dropout_4 (Dropout)          (None, 512)               0         activation_12[0][0]
__________________________________________________________________________________________________
pi (Dense)                   (None, 65)                33345     dropout_4[0][0]
__________________________________________________________________________________________________
v (Dense)                    (None, 1)                 513       dropout_4[0][0]
==================================================================================================
Total params: 26,014,274
Trainable params: 26,007,106
Non-trainable params: 7,168
```

5 Игры

5.1 Cube

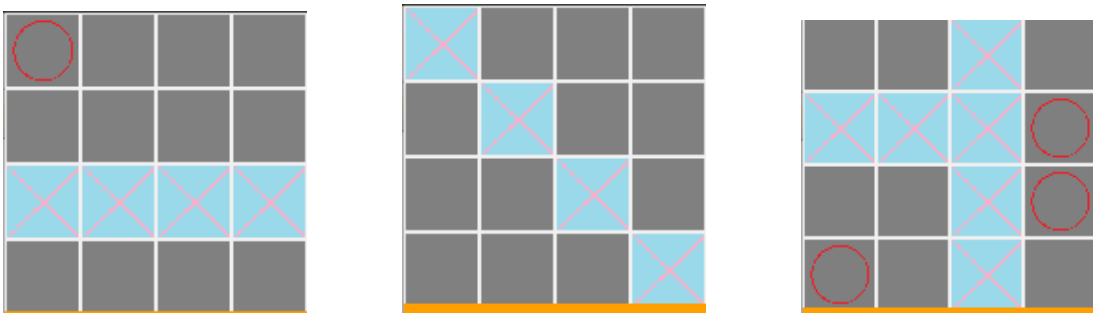
В ходе выполнения курсовой работы за основу был взят игровой фреймворк [4] со следующией структурой:

- Arena (класс, описывающий арену, в которой могут сразиться два игрока друг с другом)
- Coach (класс, описывающий процесс самостоятельной игры (4.2) нейронной сети и ее обучение)
- Game (класс, описывающий базовый класс для игры)
- MCTS (класс, описывающий поиск методом Монте - Карло в дереве (4.1))
- NeuralNet (класс, описывающий базовый класс для нейронной сети)

В данном фремворке мною была реализована антогонистическая игра *"Cube"* (крестики-нолики в трехмерном пространстве).

Рис. 3: Игра Cube

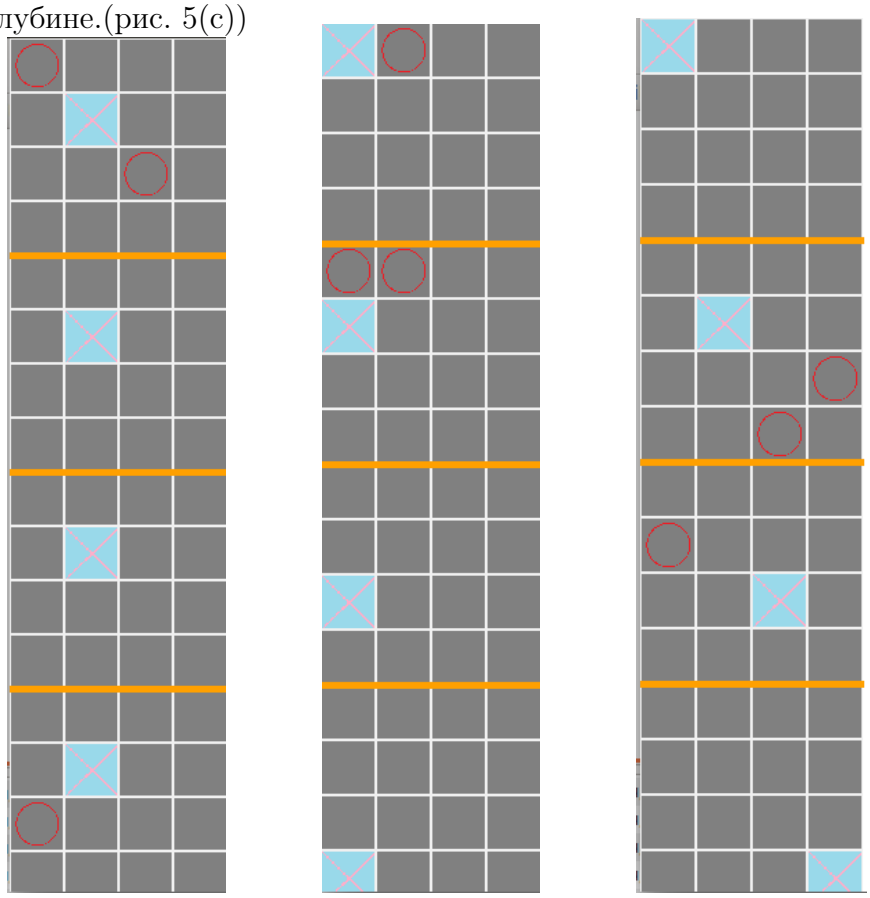
Правила "Cube" Необходимо собрать любой ряд, состоящий из 4-х одинаковых значений. Например:



(a) (b) (c)

Рис. 4: Аналогично 2-х мерным правилам

- Аналогично 2-х мерным кристикам-ноликам: на одном из слоев собрать, ряд по горизонтали, либо по вертикали, либо по диагонали.
- Собрать ряд в одном из столбцов.(рис. 5(a))
- Собрать ряд по диагонали при фиксированом значинии в линии по вертикали или по горизонтали.(рис. 5(b))
- Собрать ряд по диагонали с различными значениями по вертикали, горизонтали и глубине.(рис. 5(c))



(a) (b) (c)

Рис. 5: .

5.2 Сравнение итераций по партиям в играх

Приведем сравнение нейронной сети на разных итерациях обучения в представленной игре. Представлена статистика на основе 16 этапов обучения нейронной сети.

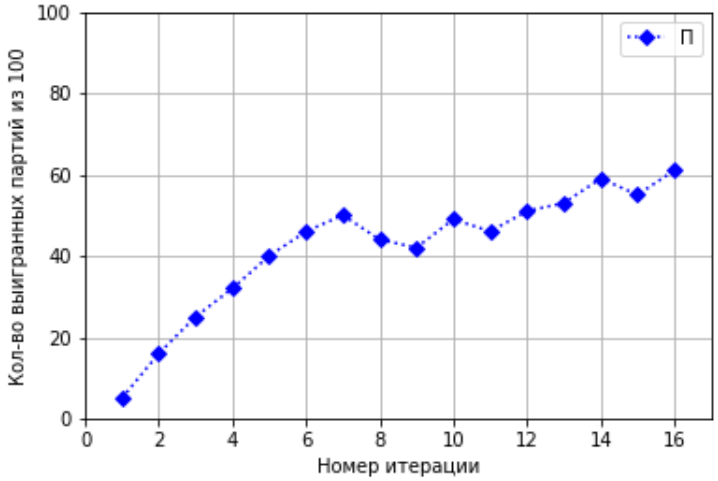


Рис. 6: Партии против osp*

osp* - Игрок. Препятсвует победе соперника, если следующий его ход победный. Выигрывает, если имеется выигрышный ход. В противном случае, ходит на рандомную позицию.

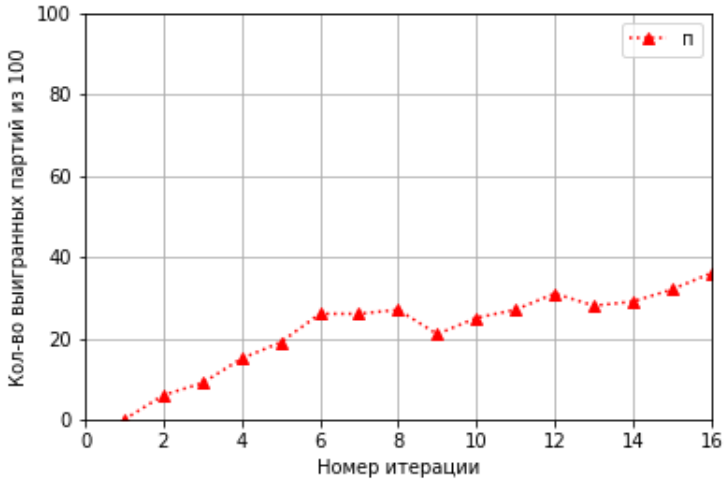


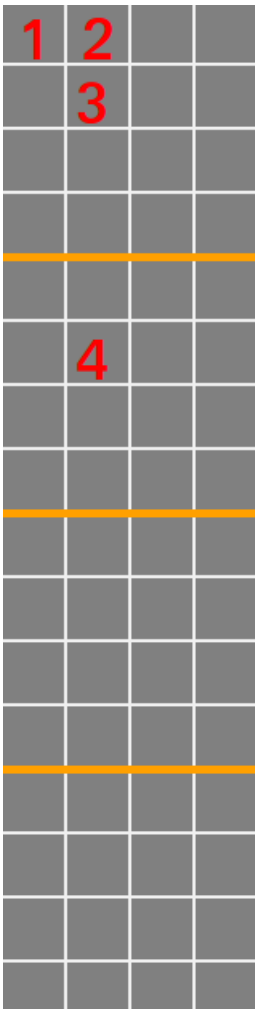
Рис. 7: Партии против hsp*

hsp* - Игрок. Препятсвует победе соперника, если следующий его ход победный. Выигрывает, если имеется выигрышный ход. В противном случае, ходит на наиболее выгодную позицию из эвристических соображений.

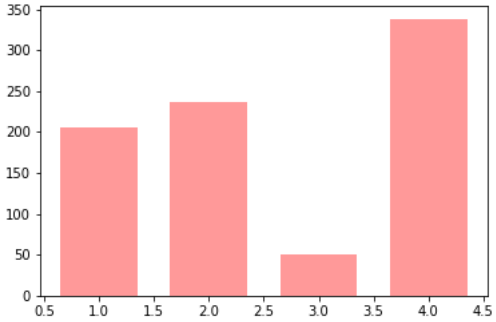
Можно заметить, что с течением времени модель нейронной сети начинает играть лучше.

5.3 Статистика начала игры

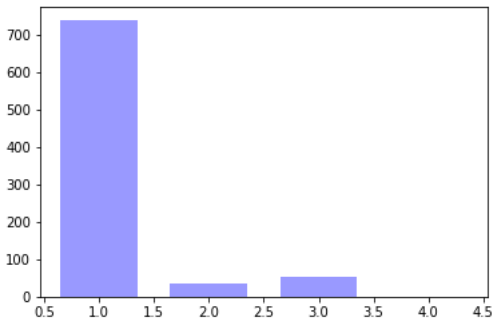
С учетом симметрий на доске, есть 4 различных вариантов ходов, которые вносят различный вклад.



a)



b)



c)

Рис. 8: a) Варианты различных ходов, b) Статистика первого хода, c) Статистика второго хода .

6 Заключение

В ходе выполнения данной работы

- Иследовали алгоритмы MCTS, UCT, UCB.

  Познакомившись с работой алгоритма MCTS и разобрав его на конкретных примерах, мы выянили, что данный алгоритм является оптимальным для поиска решений в играх с полной информацией. Алгоритм MCTS действует более эффективно, чем применяющаяся стратегия минимакса за счет сокращения ветвей древовидной структуры игры

- Иследовали применение MCTS в нейронных сетях.

- В практической части данной работы, на основе игрового фреймфорка [4] была создана игра *"Cube"* с визуальным и терминальным интерфейсами. Для данной игры получилось описать и обучить сверточную нейронную сеть. На данный момент она выигрывает 60% игр против osp*, и 35% игр против hsp*. Замечена дальнейшая тенденция к улучшению модели, но требуется больше времени для обучения.

- Провели сравнение обученных нейронных сетей на различных стадиях обучения против нескольких игроков: osp*,hsp*.

- Сделали статистику первых двух ходов с учетом симметрий на доске.

osp* - Игрок. Препятсвует победе соперника, если следующий его ход победный. Выигрывает, если имеется выигрышный ход. В противном случае, ходит на рандомную позицию.

hsp* - Игрок. Препятсвует победе соперника, если следующий его ход победный. Выигрывает, если имеется выигрышный ход. В противном случае, ходит на наиболее выгодную позицию из эвристических соображений.

Список литературы

[1] Максим Наумов, GAMBITER.RU, декабрь 2008.

[2] Владимир Попов - Заметки программистера ,DOKWORK.RU, 2012.

[3] Daniel Whitehouse - Monte Carlo Tree Search for games with Hidden Information and Uncertainty, 2014.

[4] Surag Nair - Alpha-zero-general. https://github.com/suragnair/alpha-zero-general, 2018.

[5] Surag Nair, Shantanu Thakoor, Megha Jhunjhunwala - Learning to Play Othello Without Human Knowledge , 2018.

[6] Воронцов, Машинное обучение (курс лекций, К.В.Воронцов), 2017.

[7] Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. Mastering the game of go with deep neural networks and tree search, 2016

[8] Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm, 2017

[9] Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. Mastering the game of go without human knowledge, 2017

[10] Kingma, D., and Ba, J. Adam: A method for stochastic optimization, 2014

[11] Ioffe, S., and Szegedy, C., Batch normalization: Accelerating deep network training by reducing internal covariate shift. In International Conference on Machine Learning, 448–456., 2015

[12] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting // Journal of Machine Learning Research. 2014.